
ELabELN

Developer Manual

REST API Reference, Integration Patterns, and BI/Analytics Guide

With Complete Code Examples in Python, JavaScript, and cURL

Powered by LabLynx, Inc.

www.lablynx.com | www.elabeln.com

Version 1.0 | June 2026

Table of Contents

Table of Contents	2
Preface: About This Manual	3
1. Authentication and API Basics	4
1.1 Generating an API Key	4
1.2 Request Structure	4
1.3 Authentication in Code — Three Languages	5
cURL	5
Python (requests library)	5
JavaScript (Node.js / fetch)	6
1.4 Response Format and Status Codes	6
Python error handling pattern	7
1.5 Pagination	7
Python pagination — fetch all experiments	8
2. Experiments — Complete API Reference	9
2.1 Experiment Endpoint Overview	9
2.2 Creating an Experiment	9
cURL — create and populate	10
Python — create and populate with metadata	10
2.3 Reading Experiments	11
Fetch a single experiment (full detail)	11
Python — list experiments with filtering	12
2.4 Updating Experiments (PATCH)	12
2.5 Managing Tags	13
2.6 Uploading Files (Attachments)	13
cURL file upload	14
2.7 Working with Steps	14
2.8 Linking Experiments to Resources	14
2.9 Adding Comments	15
2.10 Applying a Trusted Timestamp	15
3. Resources (Items) — Complete API Reference	16
3.1 Resource Endpoint Overview	16
3.2 Creating and Populating a Resource	16
3.3 Bulk Resource Import from CSV/Spreadsheet	17
3.4 Listing Resource Categories	18
4. Users, Teams, and System Endpoints	20
4.1 User Endpoints	20
4.2 Team Endpoints	20
4.3 Tags and Status Endpoints	21
4.4 Scheduler / Booking Endpoints	21
5. The Metadata (Custom Fields) JSON Schema	23

5.1 Schema Overview	23
5.2 All Field Types — Complete Schema Reference	23
Full metadata JSON example — HPLC run experiment	24
5.3 Reading Metadata Fields in Python	24
5.4 Writing Metadata Fields	25
6. Integration Patterns	27
6.1 Instrument Data Push — Auto-Creating Experiment Records	27
Python — HPLC post-run integration (file watcher pattern)	27
6.2 LIMS Synchronization — Linking Sample IDs	29
6.3 Writing Back Results to LIMS	30
6.4 Batch Processing — Updating Many Experiments	31
7. BI, Analytics, and Reporting	33
7.1 Extracting Data to pandas DataFrames	33
7.2 Power BI Integration — REST API Connector	34
7.3 Python — Automated QA Dashboard Report (PDF)	36
7.4 Tableau — Web Data Connector	37
8. The elabapi-python Library	40
8.1 Installation and Setup	40
8.2 Common Operations Using the Library	40
8.3 Error Handling with the Library	41
9. Webhooks and Event-Driven Integration	42
9.1 Polling for Recent Changes	42
9.2 LabVia Integration — Node-RED Based Automation	43
10. JavaScript SDK and Browser Applications	44
10.1 ELabELN JavaScript Client Module	44
10.2 Complete Node.js Integration Example	46
11. Complete API Endpoint Reference	48
11.1 All Endpoints by Resource Group	48
Experiments	48
Resources (Items)	49
Users, Teams, and System	49
12. Security, Rate Limits, and Best Practices	51
12.1 API Key Security	51
12.2 Rate Limiting	51
12.3 Handling the Read-Modify-Write Pattern Safely	52
12.4 Testing Your Integration	52
Appendix A: Environment Setup Reference	54
Python environment	54
Node.js environment	54
Required API key permissions by use case	54
Appendix B: Metadata JSON Schema Quick Reference	55

Preface: About This Manual

This manual is the complete technical reference for developers building integrations with, custom applications on top of, or BI/analytics pipelines from ELabELN. It covers:

- The ELabELN REST API v2 — every major endpoint group with explicit call signatures, request/response examples, and authentication patterns.
- The metadata (custom fields) JSON schema — how structured data is represented in the API and how to read and write it programmatically.
- Integration patterns — instrument data push, LIMS synchronization, ERP connectivity, and event-triggered automation.
- BI and analytics — extracting data for Power BI, Tableau, pandas/R analysis, and custom dashboards.
- Complete, runnable code examples in Python, JavaScript (Node.js/fetch), and cURL for every major operation.

ELabELN is built on eLabFTW, an open-source ELN platform. The REST API documented here is the eLabFTW v2 API as deployed and hosted by LabLynx. The API reference documentation (OpenAPI/Swagger specification) is available at: <https://doc.elabftw.net/api/v2/>

All code examples in this manual are production-ready with appropriate error handling, and can be used as starting points for real integrations. Variables shown in angle brackets (e.g., <YOUR_API_KEY>, <INSTANCE_URL>) must be replaced with your actual values.

1. Authentication and API Basics

1.1 Generating an API Key

Every API call must be authenticated with a personal API key. Each key is associated with the ELabELN user account that created it and carries that user's permissions. API operations executed with a key behave exactly as if that user performed the operation through the web interface.

To generate a key:

1. Log in to ELabELN.
2. Click your user avatar (top right) → Settings → API Keys tab.
3. Click Create Key. Enter a descriptive name (e.g., 'Instrument Integration — LC-MS #3').
4. Select Read-Only or Read-Write access depending on the integration's needs.
5. Copy the generated key. It is shown only once — store it in a secrets manager or environment variable, never in source code.

Note

API keys are long random strings of the form: 3-cb2314b00d2845a3f7c2891a52d8b47e... The leading number indicates the user ID. Always treat an API key as a password: rotate it periodically, use a separate key per integration, and revoke immediately if compromised.

1.2 Request Structure

The ELabELN API v2 base URL for all requests is:

```
https://<INSTANCE_URL>/api/v2/<endpoint>
```

All requests require two headers:

```
Authorization: <YOUR_API_KEY>  
Content-Type: application/json # required for POST and PATCH requests
```

HTTP verbs map directly to CRUD operations:

HTTP Verb	Operation
GET	Read data. Returns a JSON object or array. Safe and idempotent: multiple identical GETs produce the same result.
POST	Create a new resource. Returns HTTP 201 with a Location header pointing to the new resource URL.
PATCH	Update an existing resource. Partial updates: only fields included in the request body are modified. Returns HTTP 200.

DELETE

Delete a resource (soft delete in most cases). Returns HTTP 204 No Content on success.

1.3 Authentication in Code — Three Languages

cURL

```
# Set your key as an environment variable (never hardcode in scripts)
export ELAB_KEY='3-cb2314b00d2845a...'
export ELAB_URL='https://yourlab.elabeln.com'

# GET request - fetch experiment with ID 42
curl -H "Authorization: $ELAB_KEY" \
      -H "Accept: application/json" \
      "$ELAB_URL/api/v2/experiments/42"
```

Python (requests library)

```
import os
import requests

# Load credentials from environment variables
API_KEY = os.environ['ELAB_API_KEY']
BASE_URL = os.environ['ELAB_BASE_URL'] # e.g. 'https://yourlab.elabeln.com'

# Build a reusable session with auth headers
session = requests.Session()
session.headers.update({
    'Authorization': API_KEY,
    'Content-Type': 'application/json',
    'Accept': 'application/json'
})

def api_url(path):
    return f"{BASE_URL}/api/v2/{path}"

# Helper: GET with error handling
def api_get(path, params=None):
    resp = session.get(api_url(path), params=params)
    resp.raise_for_status()
    return resp.json()

# Helper: POST - returns the ID of the created resource
def api_post(path, data=None):
    resp = session.post(api_url(path), json=data or {})
    resp.raise_for_status()
    location = resp.headers.get('Location', '')
    # Extract ID from Location: .../api/v2/experiments/321
    return int(location.split('/')[-1]) if location else None

# Helper: PATCH - partial update
def api_patch(path, data):
    resp = session.patch(api_url(path), json=data)
    resp.raise_for_status()
    return resp

# Helper: DELETE
def api_delete(path):
    resp = session.delete(api_url(path))
    resp.raise_for_status()
```

```
return resp
```

JavaScript (Node.js / fetch)

```
const BASE_URL = process.env.ELAB_BASE_URL;
const API_KEY = process.env.ELAB_API_KEY;

const headers = {
  'Authorization': API_KEY,
  'Content-Type': 'application/json',
  'Accept': 'application/json'
};

async function apiGet(path, params = {}) {
  const url = new URL(`${BASE_URL}/api/v2/${path}`);
  Object.entries(params).forEach(([k, v]) => url.searchParams.set(k, v));
  const res = await fetch(url, { headers });
  if (!res.ok) throw new Error(`GET ${path} failed: ${res.status}`);
  return res.json();
}

async function apiPost(path, body = {}) {
  const res = await fetch(`${BASE_URL}/api/v2/${path}`, {
    method: 'POST', headers, body: JSON.stringify(body)
  });
  if (!res.ok) throw new Error(`POST ${path} failed: ${res.status}`);
  const location = res.headers.get('Location') || '';
  return parseInt(location.split('/').pop(), 10) || null;
}

async function apiPatch(path, body) {
  const res = await fetch(`${BASE_URL}/api/v2/${path}`, {
    method: 'PATCH', headers, body: JSON.stringify(body)
  });
  if (!res.ok) throw new Error(`PATCH ${path} failed: ${res.status}`);
  return res;
}

async function apiDelete(path) {
  const res = await fetch(`${BASE_URL}/api/v2/${path}`, {
    method: 'DELETE', headers
  });
  if (!res.ok) throw new Error(`DELETE ${path} failed: ${res.status}`);
  return res;
}
```

1.4 Response Format and Status Codes

All API responses return JSON. The standard HTTP status codes apply:

Status Code	Meaning
200 OK	Successful GET or PATCH. Response body contains the requested or updated resource.
201 Created	Successful POST. Response body may be empty; the Location header contains the URL of the new resource.
204 No Content	Successful DELETE. Empty response body.

400 Bad Request	Request body is malformed JSON or contains invalid field values. Response body contains an error message.
401 Unauthorized	API key missing, invalid, or expired.
403 Forbidden	API key is valid but lacks permission for the requested operation (e.g., read-only key attempting a write, or accessing another user's private entry).
404 Not Found	The requested resource ID does not exist or is not accessible to the API key's user.
429 Too Many Requests	Rate limit exceeded. Back off and retry after the period specified in the Retry-After header.
500 Internal Server Error	Server-side error. Retry after a brief delay; if persistent, contact LabLynx support with the request details.

Python error handling pattern

```
import requests
from requests.exceptions import HTTPError

def safe_api_get(session, url):
    try:
        resp = session.get(url)
        resp.raise_for_status()
        return resp.json()
    except HTTPError as e:
        status = e.response.status_code
        if status == 401:
            raise ValueError('Invalid API key') from e
        elif status == 403:
            raise PermissionError('Insufficient permissions') from e
        elif status == 404:
            return None # Entry not found; caller handles None
        elif status == 429:
            import time
            retry_after = int(e.response.headers.get('Retry-After', 60))
            print(f'Rate limited. Waiting {retry_after}s...')
            time.sleep(retry_after)
            return safe_api_get(session, url) # Retry once
        else:
            raise
```

1.5 Pagination

Endpoints that return lists of entries (experiments, resources, users) support pagination through query parameters. Without pagination, the API returns a default number of results (typically 15–25).

Item	Description
limit	Maximum number of results to return per page. Default varies by endpoint; maximum is typically 500.
offset	Number of records to skip before starting to collect results. Use with limit for cursor-based pagination.
q	Search/filter query string. Searches title, body text, tags, and custom fields.
cat	Filter by category ID (integer).

status	Filter by status ID (integer).
owner	Filter by user ID (integer). Returns only entries owned by this user.
extended	Set to '1' to include additional fields (e.g., full body text, metadata) in list responses. Default list responses omit large fields for performance.

Python pagination — fetch all experiments

```
def get_all_experiments(session, base_url, limit=100, **filters):
    """Generator that yields all experiments, handling pagination automatically."""
    offset = 0
    while True:
        params = {'limit': limit, 'offset': offset, **filters}
        resp = session.get(f'{base_url}/api/v2/experiments', params=params)
        resp.raise_for_status()
        data = resp.json()
        if not data: # Empty page means we've reached the end
            break
        yield from data
        if len(data) < limit:
            break # Last page returned fewer results than limit
        offset += limit

# Usage: collect all experiments in a specific category
category_id = 3 # e.g., 'HPLC Runs'
experiments = list(get_all_experiments(session, BASE_URL, cat=category_id))
print(f'Found {len(experiments)} experiments in category {category_id}')
```

2. Experiments — Complete API Reference

Experiments are the core record type in ELabELN. All experiment operations are under the /experiments path. An experiment includes: id, title, date, body (HTML), status, category, tags, metadata (custom fields), links, and attachments.

2.1 Experiment Endpoint Overview

Method	Endpoint	Description
GET	/experiments	List experiments (paginated, filterable)
POST	/experiments	Create a new empty experiment
GET	/experiments/{id}	Get full details of one experiment
PATCH	/experiments/{id}	Update experiment fields
DELETE	/experiments/{id}	Soft-delete an experiment
GET	/experiments/{id}/revisions	Get body text revision history
GET	/experiments/{id}/changelog	Get field-level change history
POST	/experiments/{id}/tags	Add a tag to an experiment
DELETE	/experiments/{id}/tags/{tag}	Remove a tag
GET	/experiments/{id}/uploads	List file attachments
POST	/experiments/{id}/uploads	Upload a file attachment
DELETE	/experiments/{id}/uploads/{uploadId}	Delete an attachment
POST	/experiments/{id}/links/items	Link to a resource (item)
POST	/experiments/{id}/links/experiments	Link to another experiment
GET	/experiments/{id}/links	Get all links from this experiment
POST	/experiments/{id}/steps	Add a step
PATCH	/experiments/{id}/steps/{stepId}	Update/complete a step
POST	/experiments/{id}/comments	Add a comment
GET	/experiments/{id}/comments	List all comments
POST	/experiments/{id}/timestamps	Apply an RFC 3161 timestamp

2.2 Creating an Experiment

Creating an experiment is a two-step process: POST to create an empty entry (which returns its ID), then PATCH to populate the content. This is intentional by the API design — the ID must be known before content is attached.

cURL — create and populate

```
# Step 1: Create empty experiment, capture the ID from the Location header
LOCATION=$(curl -sI \
  -H "Authorization: $ELAB_KEY" \
  -H "Content-Type: application/json" \
  -X POST "$ELAB_URL/api/v2/experiments" \
  | grep -i location | awk '{print $2}' | tr -d '\r')

EXP_ID=$(echo $LOCATION | grep -oP '\d+$')
echo "Created experiment ID: $EXP_ID"

# Step 2: Populate with content
curl -X PATCH \
  -H "Authorization: $ELAB_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "title": "HPLC Method Validation - Ibuprofen - 2026-06-01",
    "date": "2026-06-01",
    "body": "<h2>Objective</h2><p>Validate the HPLC method for Ibuprofen potency assay per SOP-ANAL-042.</p>",
    "status": 2,
    "category": 5,
    "tags": ["Validation", "HPLC", "Ibuprofen"]
  }' \
  "$ELAB_URL/api/v2/experiments/$EXP_ID"
```

Python — create and populate with metadata

```
import json

def create_hplc_run(session, base_url, run_data):
    """
    Create an HPLC run experiment with structured metadata.
    run_data keys: title, date, analyst, instrument_id, method,
                  lot_number, batch_id, result, passed
    """
    # Step 1: Create empty experiment
    resp = session.post(f'{base_url}/api/v2/experiments')
    resp.raise_for_status()
    exp_id = int(resp.headers['Location'].split('/')[1])

    # Build metadata JSON for custom fields
    metadata = {
        'extra_fields': {
            'Analyst': {
                'type': 'text',
                'value': run_data.get('analyst', ''),
                'position': 1
            },
            'Instrument': {
                'type': 'text',
                'value': run_data.get('instrument_id', ''),
                'position': 2
            },
            'Method Number': {
                'type': 'text',
                'value': run_data.get('method', ''),
                'position': 3
            },
            'Lot Number': {
```

```

        'type': 'text',
        'value': run_data.get('lot_number', ''),
        'position': 4
    },
    'Batch ID': {
        'type': 'text',
        'value': run_data.get('batch_id', ''),
        'position': 5
    },
    'Result (%)': {
        'type': 'number',
        'value': str(run_data.get('result', '')),
        'position': 6
    },
    'Pass / Fail': {
        'type': 'select',
        'value': 'Pass' if run_data.get('passed') else 'Fail',
        'options': ['Pass', 'Fail', 'Inconclusive'],
        'position': 7
    }
}

# Step 2: Populate all fields in one PATCH
body_html = f"<h2>Objective</h2><p>HPLC potency assay for lot
{run_data['lot_number']}</p>"
patch_data = {
    'title': run_data['title'],
    'date': run_data['date'],
    'body': body_html,
    'metadata': json.dumps(metadata),
    'tags': ['HPLC', 'Potency', run_data.get('batch_id', '')]
}
resp2 = session.patch(f'{base_url}/api/v2/experiments/{exp_id}',
json=patch_data)
resp2.raise_for_status()
print(f'Created experiment {exp_id}: {run_data["title"]}')
return exp_id

# Example usage
new_id = create_hplc_run(session, BASE_URL, {
    'title': 'HPLC Potency - Ibuprofen - Batch 4421 - 2026-06-01',
    'date': '2026-06-01',
    'analyst': 'J. Rodriguez',
    'instrument_id': 'HPLC-03',
    'method': 'SOP-ANAL-042 Rev.3',
    'lot_number': 'IBU-2026-441',
    'batch_id': '4421',
    'result': 99.3,
    'passed': True
})

```

2.3 Reading Experiments

Fetch a single experiment (full detail)

```

# cURL
curl -H "Authorization: $ELAB_KEY" \
    "$ELAB_URL/api/v2/experiments/42"

# Returns JSON with fields:
# id, elabid, title, date, body (HTML), status, category_id, tags,

```

```
# metadata (JSON string), created_at, modified_at, userid, team,
# uploads (array), links (array), steps (array), comments (array)
```

Python — list experiments with filtering

```
# Fetch all HPLC experiments that are in 'Pending Review' status
# category_id 5 = HPLC, status_id 3 = Pending Review
params = {
    'limit': 50,
    'cat': 5,      # category filter
    'status': 3,   # status filter
    'extended': 1  # include body text and metadata in list response
}
resp = session.get(f'{BASE_URL}/api/v2/experiments', params=params)
experiments = resp.json()

# Search by text query
resp = session.get(f'{BASE_URL}/api/v2/experiments', params={
    'q': 'Ibuprofen batch 4421',
    'limit': 10
})
results = resp.json()

# Fetch experiments created by a specific user
resp = session.get(f'{BASE_URL}/api/v2/experiments', params={
    'owner': 7,    # user ID
    'limit': 100
})
user_experiments = resp.json()
```

2.4 Updating Experiments (PATCH)

PATCH requests apply partial updates. Only fields included in the request body are changed; all other fields remain as-is. This makes PATCH safe for targeted updates: you can change just the status without touching the title or body.

```
# Python — update experiment status to 'Approved' (status ID 4)
resp = session.patch(
    f'{BASE_URL}/api/v2/experiments/42',
    json={'status': 4}
)
resp.raise_for_status()

# Update title only
resp = session.patch(
    f'{BASE_URL}/api/v2/experiments/42',
    json={'title': 'HPLC Potency — Ibuprofen — Batch 4421 — REVISED'}
)

# Update custom metadata fields — only the 'Result' field changes
import json

# First, fetch the current metadata to avoid overwriting other fields
current = session.get(f'{BASE_URL}/api/v2/experiments/42').json()
metadata = json.loads(current.get('metadata') or '{}')

# Modify one field
metadata.setdefault('extra_fields', {})
```

```

metadata['extra_fields']['Result (%)'] = {
    'type': 'number',
    'value': '98.7',
    'position': 6
}

# Write back
resp = session.patch(
    f'{BASE_URL}/api/v2/experiments/42',
    json={'metadata': json.dumps(metadata)}
)

```

2.5 Managing Tags

```

# Add a tag to an experiment
resp = session.post(
    f'{BASE_URL}/api/v2/experiments/42/tags',
    json={'tag': 'Approved-2026-Q2'}
)
resp.raise_for_status()

# cURL equivalent
curl -X POST \
    -H "Authorization: $ELAB_KEY" \
    -H "Content-Type: application/json" \
    -d '{"tag": "Approved-2026-Q2"}' \
    "$ELAB_URL/api/v2/experiments/42/tags"

# Remove a tag (URL-encode the tag name if it contains spaces)
curl -X DELETE \
    -H "Authorization: $ELAB_KEY" \
    "$ELAB_URL/api/v2/experiments/42/tags/OldTag"

```

2.6 Uploading Files (Attachments)

File uploads use multipart/form-data, not JSON. This is the only endpoint type in the API that does not use JSON content type.

```

# Python – upload raw instrument data file to an experiment
import os

def upload_file(session, base_url, exp_id, filepath, comment=''):
    """Upload a file attachment to an experiment."""
    with open(filepath, 'rb') as f:
        files = {'file': (os.path.basename(filepath), f,
            'application/octet-stream')}
        data = {'comment': comment} if comment else {}
        resp = session.post(
            f'{base_url}/api/v2/experiments/{exp_id}/uploads',
            files=files,
            data=data
            # Note: do NOT set Content-Type header – requests sets it automatically
            # with the correct multipart boundary when files= is used
        )
        resp.raise_for_status()
        upload_id = int(resp.headers['Location'].split('/')[-1])
        print(f'Uploaded {os.path.basename(filepath)} as upload ID {upload_id}')
        return upload_id

```

```
# Upload the instrument RAW file
upload_file(session, BASE_URL, 42,
            '/data/instruments/hplc/2026-06-01_batch4421.raw',
            comment='Agilent OpenLAB raw data file')

# Upload the peak integration report PDF
upload_file(session, BASE_URL, 42,
            '/data/instruments/hplc/2026-06-01_batch4421_report.pdf',
            comment='ChemStation peak integration report')
```

cURL file upload

```
curl -X POST \
  -H "Authorization: $ELAB_KEY" \
  -F "file=@/path/to/data.raw" \
  -F "comment=Instrument raw data file" \
  "$ELAB_URL/api/v2/experiments/42/uploads"
```

2.7 Working with Steps

```
# Python – add protocol steps to an experiment
steps = [
    'Verify instrument calibration certificate is current',
    'Prepare primary standard per SOP-REF-012',
    'Prepare test samples according to method specifications',
    'Run HPLC sequence file: SOP-ANAL-042-Seq-v3',
    'Review chromatogram and integration results',
    'Calculate potency using standard curve',
    'Complete result table and request QA review',
]

step_ids = []
for i, step_body in enumerate(steps):
    resp = session.post(
        f'{BASE_URL}/api/v2/experiments/42/steps',
        json={'body': step_body, 'ordering': i + 1}
    )
    resp.raise_for_status()
    step_id = int(resp.headers['Location'].split('/')[-1])
    step_ids.append(step_id)
    print(f'Added step {i+1}: {step_body[:50]}...')

# Mark a step as completed
resp = session.patch(
    f'{BASE_URL}/api/v2/experiments/42/steps/{step_ids[0]}',
    json={'finished': True}
)
resp.raise_for_status()
print(f'Step 1 completed')
```

2.8 Linking Experiments to Resources

```
# Link experiment 42 to resource (item) with ID 15
# (e.g., linking to the HPLC instrument resource record)

resp = session.post(
```

```

    f'{BASE_URL}/api/v2/experiments/42/links/items',
    json={'link': 15}
)
resp.raise_for_status()

# Link experiment 42 to another experiment (e.g., the parent study)
resp = session.post(
    f'{BASE_URL}/api/v2/experiments/42/links/experiments',
    json={'link': 100}    # Experiment ID 100 = parent study
)

# Get all links from experiment 42
resp = session.get(f'{BASE_URL}/api/v2/experiments/42/links')
links = resp.json()
# links contains: experiments_links[] and items_links[]
print('Linked resources:', [l['title'] for l in links.get('items_links', [])])
print('Linked experiments:', [l['title'] for l in links.get('experiments_links', [])])

```

2.9 Adding Comments

```

# Python - add a review comment to an experiment
resp = session.post(
    f'{BASE_URL}/api/v2/experiments/42/comments',
    json={'comment': 'QA review complete. Result 99.3% is within specification.',
          'comment': 'Approved for batch release. Signature will follow.'}
)
resp.raise_for_status()

# Fetch all comments on an experiment
resp = session.get(f'{BASE_URL}/api/v2/experiments/42/comments')
comments = resp.json()
for c in comments:
    print(f"[{c['created_at']}] {c['name']}: {c['comment'][:100]}")

```

2.10 Applying a Trusted Timestamp

```

# Apply an RFC 3161 trusted timestamp to an experiment
# This generates a cryptographic hash of the experiment, sends it to the
# configured TSA, and stores the timestamp token as an attachment.

resp = session.post(
    f'{BASE_URL}/api/v2/experiments/42/timestamps'
)
resp.raise_for_status()
print('Timestamp applied successfully')

# cURL equivalent
curl -X POST \
    -H "Authorization: $ELAB_KEY" \
    -H "Content-Type: application/json" \
    "$ELAB_URL/api/v2/experiments/42/timestamps"

```

3. Resources (Items) — Complete API Reference

Resources (called 'items' in the API) represent the laboratory inventory: reagents, instruments, cell lines, equipment, personnel records, and any other entity tracked in the resource catalog. The resource API mirrors the experiment API structure.

3.1 Resource Endpoint Overview

Method	Endpoint	Description
GET	/items	List resources (paginated, filterable)
POST	/items	Create a new resource
GET	/items/{id}	Get full details of one resource
PATCH	/items/{id}	Update resource fields
DELETE	/items/{id}	Delete a resource
GET	/items/{id}/uploads	List attachments
POST	/items/{id}/uploads	Upload a file to a resource
POST	/items/{id}/tags	Add a tag
DELETE	/items/{id}/tags/{tag}	Remove a tag
GET	/items/{id}/links	Get all links from this resource
POST	/items/{id}/links/items	Link to another resource
POST	/items/{id}/links/experiments	Link to an experiment
POST	/items/{id}/steps	Add a step (checklist item)
PATCH	/items/{id}/steps/{stepId}	Update/complete a step
GET	/items_types	List all resource category types

3.2 Creating and Populating a Resource

```
# Python — create a new reagent resource record

def create_reagent(session, base_url, reagent_data):
    """
    Create a reagent resource.
    reagent_data keys: title, category_id, lot_number, vendor,
                      catalog_number, expiry_date, storage_temp,
                      quantity, unit
    """
    import json

    # Create empty item
    resp = session.post(f'{base_url}/api/v2/items')
```

```

resp.raise_for_status()
item_id = int(resp.headers['Location'].split('/')[-1])

metadata = {
    'extra_fields': {
        'Vendor': {'type': 'text', 'value': reagent_data.get('vendor', '')},
        'position': 1},
        'Catalog Number': {'type': 'text', 'value':
reagent_data.get('catalog_number', ''), 'position': 2},
        'Lot Number': {'type': 'text', 'value': reagent_data.get('lot_number',
''), 'position': 3},
        'Expiry Date': {'type': 'date', 'value':
reagent_data.get('expiry_date', ''), 'position': 4},
        'Storage Temperature': {
            'type': 'select',
            'value': reagent_data.get('storage_temp', '-20°C'),
            'options': ['Room Temp', '4°C', '-20°C', '-80°C'],
            'position': 5
        },
        'Current Quantity': {
            'type': 'number',
            'value': str(reagent_data.get('quantity', '')),
            'units': ['µL', 'mL', 'L', 'mg', 'g', 'kg'],
            'unit': reagent_data.get('unit', 'mL'),
            'position': 6
        },
        'QC Status': {
            'type': 'select',
            'value': 'Pass',
            'options': ['Pass', 'Fail', 'Pending', 'Expired'],
            'position': 7
        }
    }
}

patch_data = {
    'title': reagent_data['title'],
    'category': reagent_data.get('category_id'), # items_types ID
    'metadata': json.dumps(metadata),
    'tags': ['Reagent', reagent_data.get('vendor', '')]
}

resp2 = session.patch(f'{base_url}/api/v2/items/{item_id}', json=patch_data)
resp2.raise_for_status()
return item_id

# Create an antibody reagent
ab_id = create_reagent(session, BASE_URL, {
    'title': 'Anti-p53, Clone DO-7, Lot AB-2024-441 (Abcam ab26)',
    'category_id': 2, # Category 2 = Antibodies
    'vendor': 'Abcam',
    'catalog_number': 'ab26',
    'lot_number': 'AB-2024-441',
    'expiry_date': '2026-12-31',
    'storage_temp': '-20°C',
    'quantity': 0.5,
    'unit': 'mL'
})
print(f'Created antibody resource: ID {ab_id}')

```

3.3 Bulk Resource Import from CSV/Spreadsheet

A common pattern is to import an existing inventory spreadsheet into ELabELN as resource records. The following script reads a CSV and creates one resource per row.

```

import csv
import json
import time

def bulk_import_reagents(session, base_url, csv_filepath, category_id,
                        delay_seconds=0.2):
    """
    Import reagent records from a CSV file.
    Expected CSV columns: Title, Vendor, CatalogNumber, LotNumber,
                        ExpiryDate, StorageTemp, Quantity, Unit
    Returns: list of (row_index, created_item_id) tuples
    """
    results = []
    errors = []

    with open(csv_filepath, 'r', encoding='utf-8') as f:
        reader = csv.DictReader(f)
        for i, row in enumerate(reader):
            try:
                item_id = create_reagent(session, base_url, {
                    'title': row['Title'],
                    'category_id': category_id,
                    'vendor': row.get('Vendor', ''),
                    'catalog_number': row.get('CatalogNumber', ''),
                    'lot_number': row.get('LotNumber', ''),
                    'expiry_date': row.get('ExpiryDate', ''),
                    'storage_temp': row.get('StorageTemp', ''),
                    'quantity': float(row.get('Quantity', 0) or 0),
                    'unit': row.get('Unit', 'mL')
                })
                results.append((i + 2, item_id)) # row 1 = header
                time.sleep(delay_seconds) # Be polite to the API
            except Exception as e:
                errors.append((i + 2, str(e)))
                print(f'Row {i+2} failed: {e}')

    print(f'Imported {len(results)} items. {len(errors)} errors.')
    if errors:
        print('Failed rows:', errors)
    return results

# Run import
imported = bulk_import_reagents(
    session, BASE_URL,
    'reagent_inventory_export.csv',
    category_id=2 # Antibodies
)

```

3.4 Listing Resource Categories

Before creating resources, you need the integer ID for the target category. Fetch the list of item types (resource categories) from your team:

```

# Fetch all item types (resource categories)
resp = session.get(f'{BASE_URL}/api/v2/items_types')
categories = resp.json()

# Print category name:ID mapping
print('Resource Categories:')

```

```
for cat in categories:
    print(f"  ID {cat['id']}: {cat['title']}")

# Build a lookup dict: name -> id
cat_map = {cat['title']: cat['id'] for cat in categories}
# Example: cat_map['Antibodies'] -> 2

# cURL
curl -H "Authorization: $ELAB_KEY" \
      "$ELAB_URL/api/v2/items_types"
```

4. Users, Teams, and System Endpoints

4.1 User Endpoints

Method	Endpoint	Description
GET	/users	List all users (SysAdmin only)
GET	/users/{id}	Get a specific user's profile
GET	/users/me	Get the profile of the authenticated user
PATCH	/users/{id}	Update user fields (SysAdmin only)
POST	/users	Create a new user (SysAdmin only)

```
# Get the currently authenticated user (useful to verify your key is working)
resp = session.get(f'{BASE_URL}/api/v2/users/me')
me = resp.json()
print(f"Authenticated as: {me['firstname']} {me['lastname']} (ID {me['userid']})")
print(f"Team: {me['team']}")

# Fetch all users (requires SysAdmin key)
resp = session.get(f'{BASE_URL}/api/v2/users')
users = resp.json()
print(f'Total users: {len(users)}')
active = [u for u in users if u.get('validated') == 1]
print(f'Active validated users: {len(active)}')

# Create a new user (SysAdmin only)
resp = session.post(f'{BASE_URL}/api/v2/users', json={
    'firstname': 'Jane',
    'lastname': 'Smith',
    'email': 'jane.smith@yourlab.com',
    'team': 1, # team ID
    'usergroup': 4 # role: 4 = User, 2 = Admin, 1 = SysAdmin
})
new_user_id = int(resp.headers['Location'].split('/')[-1])
print(f'Created user ID: {new_user_id}')
```

4.2 Team Endpoints

Method	Endpoint	Description
GET	/teams	List all teams
GET	/teams/{id}	Get team details
PATCH	/teams/{id}	Update team settings (Admin/SysAdmin)

```
# List all teams
resp = session.get(f'{BASE_URL}/api/v2/teams')
teams = resp.json()
```

```

for t in teams:
    print(f"ID {t['id']}: {t['name']} ({t['usercount']} users)")

# Get your current team details
resp = session.get(f'{BASE_URL}/api/v2/teams/1')
team = resp.json()
print(team)

```

4.3 Tags and Status Endpoints

Method	Endpoint	Description
GET	/tags	List all tags in current team
GET	/experiments/{id}/tags	List tags on an experiment
POST	/experiments/{id}/tags	Add a tag to an experiment
DELETE	/experiments/{id}/tags/{tag_id}	Remove a tag from an experiment
GET	/status	List experiment status values for current team

```

# Fetch all tags in the team
resp = session.get(f'{BASE_URL}/api/v2/tags')
tags = resp.json()
print(f'Total tags: {len(tags)}')
# tags is a list of {id, tag, team, created_at}
tag_map = {t['tag']: t['id'] for t in tags}

# Fetch experiment status values
resp = session.get(f'{BASE_URL}/api/v2/status')
statuses = resp.json()
status_map = {s['title']: s['id'] for s in statuses}
print('Status values:', status_map)
# Example: {'Running': 1, 'Pending Review': 2, 'Approved': 3}

```

4.4 Scheduler / Booking Endpoints

Method	Endpoint	Description
GET	/events	List all bookings (optionally filtered by resource or date)
POST	/events	Create a new booking
GET	/events/{id}	Get a specific booking
PATCH	/events/{id}	Update a booking
DELETE	/events/{id}	Cancel/delete a booking

```

# Fetch all bookings for a specific resource (item) in a date range
params = {
    'start': '2026-06-01T00:00:00',

```

```

        'end': '2026-06-30T23:59:59',
        'item': 15    # resource ID (e.g., HPLC-03 instrument)
    }
    resp = session.get(f'{BASE_URL}/api/v2/events', params=params)
    bookings = resp.json()
    print(f'{len(bookings)} bookings for resource 15 in June 2026')
    for b in bookings:
        print(f"    {b['start']} - {b['end']}: {b['title']} ({b['email']})")

# Create a new booking
resp = session.post(f'{BASE_URL}/api/v2/events', json={
    'item': 15,          # resource ID to book
    'start': '2026-06-15T09:00:00',
    'end': '2026-06-15T11:00:00',
    'title': 'Ibuprofen Method Validation Run',
})
booking_id = int(resp.headers['Location'].split('/')[-1])
print(f'Booking created: ID {booking_id}')

```

5. The Metadata (Custom Fields) JSON Schema

Custom metadata fields are stored as a JSON string in the 'metadata' field of every experiment and resource. Understanding this schema is critical for reading structured data from the API and writing it back correctly.

5.1 Schema Overview

The metadata JSON has the following top-level structure:

```
{
  "extra_fields": {
    "Field Name": {
      "type": "text|number|date|datetime-local|select|radio|checkbox|url|email",
      "value": "current value as string",
      "position": 1,
      // type-specific additional properties...
    },
    // more fields...
  },
  "elabftw": {
    "display_main_text": true,
    "extra_fields_groups": [
      {"id": 1, "name": "Group Name"}
    ]
  }
}
```

The metadata value returned by the API is a JSON-encoded string. You must parse it before reading field values and re-serialize it before writing.

5.2 All Field Types — Complete Schema Reference

type	Key Properties	Notes
text	"value": "string"	Short free text. Default type if omitted.
number	"value": "42.5", "units": ["mg","g"], "unit": "mg"	Numeric input. Value is always stored as a string.
date	"value": "2026-06-01"	ISO 8601 date (YYYY-MM-DD). Value must be in this format.
datetime-local	"value": "2026-06-01T14:30"	Date and time. Format: YYYY-MM-DDTHH:MM.
select	"value": "Option A", "options": ["Option A","Option B"]	Dropdown. Value must be one of options.
radio	"value": "Yes", "options": ["Yes","No"]	Radio group. Same structure as select but all options visible.
checkbox	"value": "on" or ""	Checked = "on", unchecked = empty string.

url	"value": "https://example.com"	URL input. Rendered as hyperlink in view mode.
email	"value": "user@lab.com"	Email address input.

Full metadata JSON example — HPLC run experiment

```
{
  "extra_fields": {
    "Analyst": {
      "type": "text",
      "value": "J. Rodriguez",
      "position": 1
    },
    "Instrument": {
      "type": "text",
      "value": "HPLC-03 (Agilent 1260)",
      "position": 2
    },
    "Run Date": {
      "type": "datetime-local",
      "value": "2026-06-01T09:15",
      "position": 3
    },
    "Sample Concentration (mg/mL)": {
      "type": "number",
      "value": "1.025",
      "position": 4
    },
    "Column Type": {
      "type": "select",
      "value": "C18 Reversed Phase",
      "options": ["C18 Reversed Phase", "C8", "HILIC", "Ion Exchange"],
      "position": 5
    },
    "Result (%)": {
      "type": "number",
      "value": "99.3",
      "position": 6
    },
    "Pass / Fail": {
      "type": "radio",
      "value": "Pass",
      "options": ["Pass", "Fail", "Inconclusive"],
      "position": 7
    },
    "Controls Included": {
      "type": "checkbox",
      "value": "on",
      "position": 8
    },
    "Raw Data Location": {
      "type": "url",
      "value": "https://labdrive.yourlab.com/HPLC/2026/batch4421.raw",
      "position": 9
    }
  }
}
```

5.3 Reading Metadata Fields in Python

```

import json

def get_metadata_fields(experiment):
    """
    Parse metadata from an experiment dict returned by the API.
    Returns a dict: field_name -> value (as string)
    """
    raw = experiment.get('metadata') or '{}'
    meta = json.loads(raw)
    fields = meta.get('extra_fields', {})
    return {name: field.get('value', '') for name, field in fields.items()}

# Example: extract result and pass/fail from a QA experiment
exp = session.get(f'{BASE_URL}/api/v2/experiments/42').json()
fields = get_metadata_fields(exp)

result_pct = float(fields.get('Result (%)', 0))
passed = fields.get('Pass / Fail', '') == 'Pass'
analyst = fields.get('Analyst', 'Unknown')

print(f'Analyst: {analyst}')
print(f'Result: {result_pct:.1f}% → {"PASS" if passed else "FAIL"}')

# Extract with type information (including units)
def get_metadata_full(experiment):
    """Returns the full extra_fields dict with all properties."""
    raw = experiment.get('metadata') or '{}'
    return json.loads(raw).get('extra_fields', {})

fields_full = get_metadata_full(exp)
concentration_field = fields_full.get('Sample Concentration (mg/mL)', {})
concentration = concentration_field.get('value', '')
unit = concentration_field.get('unit', '') # from units dropdown
print(f'Concentration: {concentration} {unit}')

```

5.4 Writing Metadata Fields

When writing metadata back to ELabELN, always use the read-modify-write pattern: fetch the current metadata, modify only the target field, and patch it back. This prevents accidentally erasing fields you didn't intend to modify.

```

import json

def update_metadata_field(session, base_url, entity_type, entity_id,
                          field_name, new_value):
    """
    Update a single custom metadata field without touching other fields.
    entity_type: 'experiments' or 'items'
    """
    # 1. Fetch current state
    resp = session.get(f'{base_url}/api/v2/{entity_type}/{entity_id}')
    resp.raise_for_status()
    current = resp.json()

    # 2. Parse existing metadata
    raw = current.get('metadata') or '{}'
    meta = json.loads(raw)
    meta.setdefault('extra_fields', {})

    # 3. Update the target field's value only
    if field_name in meta['extra_fields']:

```

```
        meta['extra_fields'][field_name]['value'] = str(new_value)
    else:
        # Field doesn't exist yet; create it as text type
        meta['extra_fields'][field_name] = {'type': 'text', 'value':
str(new_value)}

    # 4. Write back
    resp2 = session.patch(
        f'{base_url}/api/v2/{entity_type}/{entity_id}',
        json={'metadata': json.dumps(meta)}
    )
    resp2.raise_for_status()

# Update a single field on experiment 42
update_metadata_field(session, BASE_URL, 'experiments', 42,
                      'Pass / Fail', 'Pass')

# Update the current quantity on a resource record
update_metadata_field(session, BASE_URL, 'items', 15,
                      'Current Quantity', '0.25')
```

6. Integration Patterns

This chapter covers the most common integration architectures for ELabELN: instrument data push, LIMS synchronization, ERP/procurement connectivity, and event-triggered automation. Each pattern includes a complete, runnable example.

6.1 Instrument Data Push — Auto-Creating Experiment Records

The most powerful automation pattern is having laboratory instruments (or their data acquisition software) automatically create ELabELN experiment records at the end of each run. This eliminates manual transcription, ensures contemporaneous documentation, and attaches raw data files automatically.

The integration hook — where to call the ELabELN API — depends on the instrument software:

- **Post-run script:** Many instruments (Agilent OpenLAB, Waters Empower, Thermo Xcalibur) support scripting that runs after a sequence completes.
- **File watcher:** A Python daemon watches the instrument's data output folder and triggers on new files.
- **LabVia integration:** LabLynx's LabVia (Node-RED) provides a visual flow builder for instrument connectivity, including pre-built connectors for many common instruments.

Python — HPLC post-run integration (file watcher pattern)

```
"""
hplc_watcher.py
Watches a folder for new HPLC .raw files and auto-creates ELabELN experiments.
Run as a background service on the instrument PC or a network share monitor.
"""
import os
import json
import time
import requests
from pathlib import Path
from datetime import datetime
import logging

logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s %(message)s')
log = logging.getLogger('hplc_watcher')

# Configuration - load from environment in production
API_KEY = os.environ['ELAB_API_KEY']
BASE_URL = os.environ['ELAB_BASE_URL']
WATCH_DIR = os.environ.get('HPLC_OUTPUT_DIR', '/data/hplc/output')
CATEGORY_ID = int(os.environ.get('HPLC_CATEGORY_ID', '5')) # HPLC Runs
INSTRUMENT_RESOURCE_ID = int(os.environ.get('HPLC_RESOURCE_ID', '12'))

session = requests.Session()
session.headers.update({
    'Authorization': API_KEY,
    'Content-Type': 'application/json'
})

def parse_run_info(raw_filepath):
    """
```

```

Extract run metadata from the .raw file path / companion .txt file.
Adjust parsing logic to match your instrument software's output format.
"""
stem = Path(raw_filepath).stem # e.g. '2026-06-01_Batch4421_Analyst-JR'
parts = stem.split('_')
return {
    'date': parts[0] if parts else datetime.now().strftime('%Y-%m-%d'),
    'batch': parts[1] if len(parts) > 1 else '',
    'analyst': parts[2].replace('Analyst-', '') if len(parts) > 2 else '',
    'filename': Path(raw_filepath).name
}

def create_experiment_for_run(filepath):
    """Create an ELabELN experiment record for a completed HPLC run."""
    info = parse_run_info(filepath)
    today = datetime.now().strftime('%Y-%m-%d')

    # 1. Create experiment
    resp = session.post(f'{BASE_URL}/api/v2/experiments')
    resp.raise_for_status()
    exp_id = int(resp.headers['Location'].split('/')[-1])

    # 2. Build metadata
    metadata = {
        'extra_fields': {
            'Analyst': {'type': 'text', 'value': info['analyst'], 'position': 1},
            'Batch ID': {'type': 'text', 'value': info['batch'], 'position': 2},
            'Run Date': {'type': 'date', 'value': info['date'], 'position': 3},
            'Raw File': {'type': 'text', 'value': info['filename'], 'position': 4},
            'Integration Status': {
                'type': 'select',
                'value': 'Auto-imported',
                'options': ['Auto-imported', 'Verified', 'Needs Review'],
                'position': 5
            }
        }
    }

    # 3. Populate experiment
    session.patch(f'{BASE_URL}/api/v2/experiments/{exp_id}', json={
        'title': f'HPLC Run - {info["batch"]} - {info["date"]} (Auto)',
        'date': info['date'],
        'category': CATEGORY_ID,
        'metadata': json.dumps(metadata),
        'tags': ['HPLC', 'Auto-Imported', info['batch']]
    }).raise_for_status()

    # 4. Link to instrument resource
    session.post(f'{BASE_URL}/api/v2/experiments/{exp_id}/links/items',
                json={'link': INSTRUMENT_RESOURCE_ID}).raise_for_status()

    # 5. Upload raw data file
    with open(filepath, 'rb') as f:
        session.post(f'{BASE_URL}/api/v2/experiments/{exp_id}/uploads',
                    files={'file': (info['filename'], f,
                                   'application/octet-stream')},
                    data={'comment': 'Instrument raw data
(auto-attached)'}).raise_for_status()

    log.info(f'Created experiment {exp_id} for {info["filename"]}')
    return exp_id

def watch_folder():
    """Poll for new .raw files and process them."""

```

```

seen = set(Path(WATCH_DIR).glob('*.raw')) # seed with existing files
log.info(f'Watching {WATCH_DIR} for new .raw files...')
while True:
    current = set(Path(WATCH_DIR).glob('*.raw'))
    new_files = current - seen
    for f in sorted(new_files):
        log.info(f'New file detected: {f.name}')
        try:
            create_experiment_for_run(str(f))
        except Exception as e:
            log.error(f'Failed to process {f.name}: {e}')
    seen = current
    time.sleep(30) # Poll every 30 seconds

if __name__ == '__main__':
    watch_folder()

```

6.2 LIMS Synchronization — Linking Sample IDs

When ELabELN is used alongside a LIMS (such as ELab LIMS), you need to link LIMS sample accession numbers to ELabELN experiment records. The Custom ID and metadata fields enable this cross-system reference.

```

"""
lims_sync.py
Synchronizes sample data from ELab LIMS to ELabELN experiments.
Polls the LIMS API for newly received samples and creates
corresponding ELabELN experiment records.
"""
import os, json, requests
from datetime import datetime

ELAB_KEY = os.environ['ELAB_API_KEY']
ELAB_URL = os.environ['ELAB_BASE_URL']
LIMS_KEY = os.environ['LIMS_API_KEY']
LIMS_URL = os.environ['LIMS_BASE_URL']

elab = requests.Session()
elab.headers.update({'Authorization': ELAB_KEY, 'Content-Type':
'application/json'})

lims = requests.Session()
lims.headers.update({'Authorization': f'Bearer {LIMS_KEY}', 'Content-Type':
'application/json'})

def sync_sample_to_eln(sample):
    """
    Given a LIMS sample dict, create an ELabELN experiment pre-populated
    with the LIMS accession number and sample metadata.
    """
    accession = sample['accession_number'] # e.g. 'ACC-2026-00412'

    # Check if already synced (search by Custom ID / accession in title)
    existing = elab.get(f'{ELAB_URL}/api/v2/experiments',
        params={'q': accession, 'limit': 1}).json()

    if existing:
        print(f'{accession} already in ELabELN as experiment {existing[0]["id"]}')
        return existing[0]['id']

    # Create new experiment

```

```

resp = elab.post(f'{ELAB_URL}/api/v2/experiments')
resp.raise_for_status()
exp_id = int(resp.headers['Location'].split('/')[-1])

metadata = {
    'extra_fields': {
        'LIMS Accession Number': {
            'type': 'text', 'value': accession, 'position': 1
        },
        'Sample Type': {
            'type': 'text', 'value': sample.get('sample_type', ''), 'position':
2
        },
        'Received Date': {
            'type': 'date', 'value': sample.get('received_date', ''),
'position': 3
        },
        'Client': {
            'type': 'text', 'value': sample.get('client_name', ''), 'position':
4
        },
        'Tests Requested': {
            'type': 'text', 'value': ', '.join(sample.get('tests', [])),
'position': 5
        },
        'Priority': {
            'type': 'select',
            'value': sample.get('priority', 'Routine'),
            'options': ['STAT', 'Priority', 'Routine'],
            'position': 6
        },
        'LIMS Sync Status': {
            'type': 'select',
            'value': 'Synced',
            'options': ['Synced', 'Updated', 'Error'],
            'position': 7
        }
    }
}

elab.patch(f'{ELAB_URL}/api/v2/experiments/{exp_id}', json={
    'title': f'Sample Analysis - {accession} - {sample.get("sample_type", "")}',
    'date': sample.get('received_date', datetime.now().strftime('%Y-%m-%d')),
    'metadata': json.dumps(metadata),
    'tags': ['LIMS-Synced', sample.get('sample_type', ''),
            sample.get('priority', 'Routine')]
}).raise_for_status()

print(f'Created ELabELN experiment {exp_id} for LIMS sample {accession}')
return exp_id

```

6.3 Writing Back Results to LIMS

After analysts complete their experiments in ELabELN, your integration can extract results from metadata fields and POST them back to the LIMS to close the sample workflow.

```

def sync_results_back_to_lims(exp_id):
    """
    Read result from ELabELN experiment metadata and update LIMS sample record.
    """
    import json

```

```

# 1. Fetch experiment
exp = elab.get(f'{ELAB_URL}/api/v2/experiments/{exp_id}').json()
fields = json.loads(exp.get('metadata') or '{}').get('extra_fields', {})

# 2. Extract key fields
accession = fields.get('LIMS Accession Number', {}).get('value', '')
result_pct = fields.get('Result (%)', {}).get('value', '')
pass_fail = fields.get('Pass / Fail', {}).get('value', '')
analyst = fields.get('Analyst', {}).get('value', '')

if not accession:
    print(f'Experiment {exp_id} has no LIMS accession; skipping sync.')
    return

# 3. Check experiment is Approved (status change triggers the sync)
status_id = exp.get('status')
if status_id != 4: # 4 = Approved
    print(f'Experiment {exp_id} not yet approved; not syncing.')
    return

# 4. POST results to LIMS
lims_payload = {
    'accession': accession,
    'result_value': result_pct,
    'result_status': pass_fail,
    'analyst': analyst,
    'approved_date': datetime.now().strftime('%Y-%m-%d'),
    'eln_experiment_id': exp_id
}
resp = lims.patch(f'{LIMS_URL}/api/samples/{accession}/results',
                  json=lims_payload)
resp.raise_for_status()
print(f'Updated LIMS sample {accession} with result {result_pct}%
      ({pass_fail})')

```

6.4 Batch Processing — Updating Many Experiments

When processing large numbers of experiments (e.g., batch-updating status after a release decision, or applying a new tag to all experiments in a project), use the API with rate limiting to avoid overwhelming the server.

```

import time

def batch_update_status(session, base_url, exp_ids, new_status_id,
                        delay=0.1, dry_run=False):
    """
    Update the status of a list of experiments.
    delay: seconds between requests (be polite to the API)
    dry_run: if True, print what would be done without making changes
    """
    success, failed = [], []
    for exp_id in exp_ids:
        if dry_run:
            print(f'[DRY RUN] Would set experiment {exp_id} to status
                  {new_status_id}')
            continue
        try:
            resp = session.patch(
                f'{base_url}/api/v2/experiments/{exp_id}',
                json={'status': new_status_id}
            )

```

```
        resp.raise_for_status()
        success.append(exp_id)
    except Exception as e:
        print(f'Failed for experiment {exp_id}: {e}')
        failed.append(exp_id)
    time.sleep(delay)

    print(f'Updated {len(success)}/{len(exp_ids)} experiments.')
    if failed:
        print(f'Failed IDs: {failed}')
    return success, failed

# Usage: mark all HPLC experiments in a date range as 'Archived'
from datetime import date

# Collect experiment IDs for HPLC runs in Q1 2026
all_hplc = list(get_all_experiments(session, BASE_URL,
                                   cat=5, # HPLC
                                   status=4)) # Approved
q1_ids = [e['id'] for e in all_hplc
          if '2026-01' <= e['date'] <= '2026-03-31']

# Dry run first
batch_update_status(session, BASE_URL, q1_ids, 6, dry_run=True)

# If it looks right, execute
batch_update_status(session, BASE_URL, q1_ids, 6) # 6 = Archived
```

7. BI, Analytics, and Reporting

ELabELN's API makes it straightforward to extract structured data for business intelligence, analytics dashboards, and automated reporting. This chapter covers data extraction patterns for Python/pandas, Power BI, and Tableau, plus a custom report generation example.

7.1 Extracting Data to pandas DataFrames

The most flexible analytics approach is to extract experiment data via the API into pandas DataFrames for analysis in Python. This pattern is suitable for ad-hoc analysis, Jupyter notebook workflows, and building pipelines that feed downstream visualization tools.

```
"""
el_n_analytics.py
Extract ELabELN experiment data into pandas for analysis and reporting.
"""

import os, json, requests
import pandas as pd
from datetime import datetime

API_KEY = os.environ['ELAB_API_KEY']
BASE_URL = os.environ['ELAB_BASE_URL']

session = requests.Session()
session.headers.update({'Authorization': API_KEY})

def fetch_experiments_to_df(category_id=None, status_id=None,
                           date_from=None, date_to=None,
                           include_metadata=True):
    """
    Fetch experiments and return a flat pandas DataFrame.
    Metadata custom fields are expanded into individual columns.
    """
    params = {'limit': 500, 'extended': 1}
    if category_id: params['cat'] = category_id
    if status_id: params['status'] = status_id

    # Paginate through all results
    all_exps = []
    offset = 0
    while True:
        params['offset'] = offset
        resp = session.get(f'{BASE_URL}/api/v2/experiments', params=params)
        resp.raise_for_status()
        page = resp.json()
        if not page: break
        all_exps.extend(page)
        if len(page) < params['limit']: break
        offset += params['limit']

    if not all_exps:
        return pd.DataFrame()

    rows = []
    for exp in all_exps:
        row = {
            'id': exp.get('id'),
```

```

        'title': exp.get('title', ''),
        'date': pd.to_datetime(exp.get('date'), errors='coerce'),
        'status_id': exp.get('status'),
        'category_id': exp.get('category_id'),
        'user_id': exp.get('userid'),
        'created_at': pd.to_datetime(exp.get('created_at'), errors='coerce'),
        'modified_at': pd.to_datetime(exp.get('modified_at'), errors='coerce'),
        'tags': ','.join(t.get('tag', '') for t in exp.get('tags', [])),
        'has_uploads': len(exp.get('uploads', [])) > 0,
        'upload_count': len(exp.get('uploads', [])),
        'link_count': len(exp.get('links', {}).get('items_links', [])),
    }

    # Expand metadata fields as columns
    if include_metadata and exp.get('metadata'):
        try:
            meta = json.loads(exp['metadata'])
            for fname, fdata in meta.get('extra_fields', {}).items():
                col = f'field_{fname.lower().replace(" ", "_")}'
                row[col] = fdata.get('value', '')
        except (json.JSONDecodeError, AttributeError):
            pass

    rows.append(row)

df = pd.DataFrame(rows)
if 'date' in df.columns:
    df = df.sort_values('date', ascending=False)
return df

# Example: HPLC QA analytics
df = fetch_experiments_to_df(category_id=5) # HPLC Runs

# Cast numeric metadata field to float
if 'field_result_(%)' in df.columns:
    df['result_pct'] = pd.to_numeric(df['field_result_(%)'], errors='coerce')

# Calculate pass rate by month
df['month'] = df['date'].dt.to_period('M')
if 'field_pass_/fail' in df.columns:
    monthly = df.groupby('month').agg(
        total_runs=('id', 'count'),
        pass_count=('field_pass_/fail', lambda x: (x == 'Pass').sum()),
        mean_result=('result_pct', 'mean')
    )
    monthly['pass_rate'] = (monthly['pass_count'] / monthly['total_runs'] *
100).round(1)
    print(monthly.to_string())

```

7.2 Power BI Integration — REST API Connector

Power BI can connect directly to the ELabELN API using the Power Query Web connector with custom authentication headers. The following Power Query M script fetches experiment data for a specific category.

```

// Power BI Desktop — Power Query M script
// To use: Data → Get Data → Blank Query → paste this in the Advanced Editor

let
    // Configuration
    BaseUrl = "https://yourlab.elabeln.com",

```

```

ApiKey = "3-YOUR_API_KEY_HERE",
CategoryId = "5", // Change to your category ID

// Fetch page function
FetchPage = (offset as number) as list =>
    let
        Url = BaseUrl & "/api/v2/experiments?limit=200&offset="
            & Number.ToText(offset) & "&cat=" & CategoryId & "&extended=1",
        Response = Web.Contents(Url, [
            Headers = [
                Authorization = ApiKey,
                #"Content-Type" = "application/json"
            ]
        ]),
        Json = Json.Document(Response)
    in
        if Json = null then {} else Json,

// Paginate – fetch up to 10 pages (2000 records)
Pages = List.Generate(
    () => [page = FetchPage(0), offset = 0],
    each List.Count([page]) > 0,
    each [page = FetchPage([offset] + 200), offset = [offset] + 200],
    each [page]
),
AllRecords = List.Combine(Pages),

// Convert to table
Table = Table.FromList(AllRecords, Splitter.SplitByNothing()),
Expanded = Table.ExpandRecordColumn(Table, "Column1",
    {"id", "title", "date", "status", "category_id",
     "metadata", "created_at", "modified_at"},
    {"ID", "Title", "Date", "Status", "CategoryID",
     "Metadata", "Created", "Modified"}
),

// Parse metadata to extract custom fields
WithMeta = Table.AddColumn(Expanded, "MetaParsed",
    each if [Metadata] = null then null else
        try Json.Document([Metadata]) otherwise null
),
// Extract specific fields from metadata
WithAnalyst = Table.AddColumn(WithMeta, "Analyst",
    each try [MetaParsed][extra_fields][Analyst][value] otherwise ""
),
WithResult = Table.AddColumn(WithAnalyst, "Result_Pct",
    each try Number.From([MetaParsed][extra_fields][#"Result (%)"])[value]
otherwise null
),
WithPassFail = Table.AddColumn(WithResult, "Pass_Fail",
    each try [MetaParsed][extra_fields][#"Pass / Fail"])[value] otherwise ""
),

// Clean up
FinalTable = Table.RemoveColumns(WithPassFail, {"Metadata", "MetaParsed"}),
TypedTable = Table.TransformColumnTypes(FinalTable, {
    {"Date", type date}, {"Created", type datetime},
    {"Result_Pct", type number}
})
in
TypedTable

```

Tip

In Power BI Service (cloud), you need a Data Gateway to connect to on-premises ELabELN instances. For sciCloud.net® (cloud) deployments, the API is directly accessible from Power BI Service without a gateway.

7.3 Python — Automated QA Dashboard Report (PDF)

This example generates a monthly QA summary report as a formatted PDF using the ELabELN API data and matplotlib/reportlab.

```
"""
qa_report.py
Generates a monthly QA summary report from ELabELN data.
Requires: pandas, matplotlib, requests
"""

import json, io, os
from datetime import datetime, timedelta
from calendar import month_name
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
import requests

API_KEY = os.environ['ELAB_API_KEY']
BASE_URL = os.environ['ELAB_BASE_URL']

session = requests.Session()
session.headers.update({'Authorization': API_KEY})

def get_qa_data(year, month):
    """Fetch all QA test results for a given month."""
    # Calculate date range
    start = f'{year}-{month:02d}-01'
    import calendar
    last_day = calendar.monthrange(year, month)[1]
    end = f'{year}-{month:02d}-{last_day}'

    params = {'limit': 500, 'cat': 5, 'extended': 1}
    resp = session.get(f'{BASE_URL}/api/v2/experiments', params=params)
    exps = [e for e in resp.json()
            if start <= e.get('date', '') <= end]

    rows = []
    for e in exps:
        meta_fields = {}
        if e.get('metadata'):
            try:
                meta_fields = {k: v.get('value', '') for k, v in
                                json.loads(e['metadata']).get('extra_fields', {}).items()}
            except: pass
        rows.append({
            'date': e['date'],
            'analyst': meta_fields.get('Analyst', ''),
            'result': pd.to_numeric(meta_fields.get('Result (%)', '')),
            'pass_fail': meta_fields.get('Pass / Fail', ''),
            'instrument': meta_fields.get('Instrument', ''),
            'batch': meta_fields.get('Batch ID', ''),
        })
    return pd.DataFrame(rows)
```

```

def generate_report(year, month, output_path):
    df = get_qa_data(year, month)
    if df.empty:
        print('No data for this period.')
        return

    fig, axes = plt.subplots(2, 2, figsize=(12, 8))
    fig.suptitle(f'QA Summary - {month_name[month]} {year}', fontsize=16,
fontweight='bold')

    # Plot 1: Pass/Fail pie chart
    pf = df['pass_fail'].value_counts()
    axes[0,0].pie(pf, labels=pf.index, autopct='%1.1f%%',
        colors=['#2E7D32', '#B71C1C', '#E65100'])
    axes[0,0].set_title('Pass / Fail Distribution')

    # Plot 2: Results trend over month
    df_sorted = df.sort_values('date')
    axes[0,1].plot(range(len(df_sorted)), df_sorted['result'], 'o-',
color='#1F3864')
    axes[0,1].axhline(y=97.0, color='red', linestyle='--', label='Lower spec
(97%)')
    axes[0,1].axhline(y=103.0, color='red', linestyle='--', label='Upper spec
(103%)')
    axes[0,1].set_title('Result (%) - Chronological')
    axes[0,1].set_ylabel('Potency (%)')
    axes[0,1].legend(fontsize=8)

    # Plot 3: Runs per analyst
    analyst_counts = df['analyst'].value_counts()
    axes[1,0].barh(analyst_counts.index, analyst_counts.values, color='#1F3864')
    axes[1,0].set_title('Runs by Analyst')
    axes[1,0].set_xlabel('Number of Runs')

    # Plot 4: Runs per instrument
    instr_counts = df['instrument'].value_counts()
    axes[1,1].bar(instr_counts.index, instr_counts.values, color='#E8622A')
    axes[1,1].set_title('Runs by Instrument')
    axes[1,1].set_ylabel('Number of Runs')
    plt.setp(axes[1,1].xaxis.get_majorticklabels(), rotation=45, ha='right')

    # Stats box
    stats = (
        f'Total runs: {len(df)}\n'
        f'Pass rate: {(df["pass_fail"]=="Pass").mean()*100:.1f}%\n'
        f'Mean result: {df["result"].mean():.2f}%\n'
        f'Std dev: {df["result"].std():.2f}%\n'
    )
    fig.text(0.01, 0.01, stats, fontsize=9, va='bottom',
        bbox=dict(boxstyle='round', facecolor='lightyellow', alpha=0.8))

    plt.tight_layout()
    plt.savefig(output_path, dpi=150, bbox_inches='tight')
    plt.close()
    print(f'Report saved: {output_path}')

# Generate June 2026 report
generate_report(2026, 6, 'qa_report_2026-06.pdf')

```

7.4 Tableau — Web Data Connector

Tableau can connect to REST APIs via Web Data Connectors (WDC). The following JavaScript snippet is the core of a WDC that fetches E LabELN experiments.

```
// Tableau Web Data Connector (WDC) - E LabELN Experiments
// Save as index.html and host on a web server accessible from Tableau Desktop

(function() {
  var myConnector = tableau.makeConnector();

  myConnector.getSchema = function(schemaCallback) {
    var cols = [
      {id:'id',      alias:'Experiment ID',  dataType:tableau.dataTypeEnum.int},
      {id:'title',   alias:'Title',         dataType:tableau.dataTypeEnum.string},
      {id:'date',    alias:'Date',          dataType:tableau.dataTypeEnum.date},
      {id:'status',  alias:'Status ID',      dataType:tableau.dataTypeEnum.int},
      {id:'analyst', alias:'Analyst',        dataType:tableau.dataTypeEnum.string},
      {id:'result',  alias:'Result (%)',     dataType:tableau.dataTypeEnum.float},
      {id:'passfail', alias:'Pass/Fail',     dataType:tableau.dataTypeEnum.string},
      {id:'tags',    alias:'Tags',          dataType:tableau.dataTypeEnum.string},
    ];
    var tableSchema = {
      id: 'elabeln_experiments',
      alias: 'E LabELN Experiments',
      columns: cols
    };
    schemaCallback([tableSchema]);
  };

  myConnector.getData = function(table, doneCallback) {
    var conn = JSON.parse(tableau.connectionData);
    var baseUrl = conn.baseUrl;
    var apiKey = conn.apiKey;
    var catId = conn.categoryId;

    function fetchPage(offset, accumulated) {
      $.ajax({
        url: baseUrl + '/api/v2/experiments?limit=200&offset=' + offset
          + '&cat=' + catId + '&extended=1',
        headers: { 'Authorization': apiKey },
        success: function(data) {
          var rows = data.map(function(exp) {
            var fields = {};
            try {
              fields = JSON.parse(exp.metadata || '{}').extra_fields || {};
            } catch(e) {}
            return {
              id: exp.id,
              title: exp.title || '',
              date: (exp.date || '').substring(0,10),
              status: exp.status || 0,
              analyst: (fields['Analyst'] || {}).value || '',
              result: parseFloat((fields['Result (%)'] || {}).value) || null,
              passfail: (fields['Pass / Fail'] || {}).value || '',
              tags: (exp.tags || []).map(t => t.tag).join(','),
            };
          });
          accumulated = accumulated.concat(rows);
          if (data.length === 200) {
            fetchPage(offset + 200, accumulated); // next page
          }
        }
      });
    }
  };
});
```

```
        } else {
            table.appendRows(accumulated);
            doneCallback();
        }
    },
    error: function(xhr) {
        tableau.abortWithError('API error: ' + xhr.status);
    }
});
}
fetchPage(0, []);
};

tableau.registerConnector(myConnector);
})();
```

8. The elabapi-python Library

The official Python client library for the ELabELN API (elabapi-python) provides a generated, typed interface to all API endpoints. It is the recommended approach for complex Python integrations because it handles authentication, request construction, and response parsing automatically.

8.1 Installation and Setup

```
# Install the Python client library
pip install elabapi-python

# Or from GitHub (for the latest version)
pip install git+https://github.com/elabftw/elabapi-python.git
```

```
# Python - configure the client
import elabapi_python

configuration = elabapi_python.Configuration()
configuration.api_key['token'] = 'YOUR_API_KEY'
configuration.api_key_prefix['token'] = '' # no prefix
configuration.host = 'https://yourlab.elabeln.com/api/v2'
configuration.debug = False # set True for request/response logging

# Create API client
api_client = elabapi_python.ApiClient(configuration)

# Instantiate specific API objects
experiments_api = elabapi_python.ExperimentsApi(api_client)
items_api = elabapi_python.ItemsApi(api_client)
users_api = elabapi_python.UsersApi(api_client)
tags_api = elabapi_python.TagsApi(api_client)
events_api = elabapi_python.EventsApi(api_client)
```

8.2 Common Operations Using the Library

```
# List experiments
experiments = experiments_api.read_experiments(
    limit=50,
    category=5, # HPLC category
    status=4    # Approved
)
for exp in experiments:
    print(f'{exp.id}: {exp.title} - {exp.date}')

# Get a single experiment
exp = experiments_api.get_experiment(42)
print(exp.title, exp.date)

# Create an experiment
created = experiments_api.create_experiment(body={})
new_id = int(created.headers['Location'].split('/')[-1])

# Patch (update) an experiment
experiments_api.patch_experiment(
```

```
new_id,
body={
    'title': 'My new experiment',
    'date': '2026-06-01',
    'status': 1
}
)

# Add a tag
experiments_api.create_experiment_tag(new_id, body={'tag': 'ProjectAlpha'})

# Upload a file
import io
with open('data.raw', 'rb') as f:
    experiments_api.create_experiment_upload(
        new_id,
        file=f,
        comment='Raw instrument output'
    )

# Read items (resources)
items = items_api.read_items(limit=100, category=2)
for item in items:
    print(f'{item.id}: {item.title}')

# Get current user
me = users_api.read_user('me')
print(f'Logged in as: {me.firstname} {me.lastname}')
```

8.3 Error Handling with the Library

```
from elabapi_python.rest import ApiException

try:
    exp = experiments_api.get_experiment(99999)
except ApiException as e:
    if e.status == 404:
        print('Experiment not found')
    elif e.status == 403:
        print('No permission to access this experiment')
    else:
        print(f'API error {e.status}: {e.body}')
```

9. Webhooks and Event-Driven Integration

ELabELN does not provide built-in webhook delivery (push notifications when things change). Event-driven integration is achieved through polling the API for recent changes, or through LabVia (the LabLynx integration platform, built on Node-RED) which supports event-triggered workflows.

9.1 Polling for Recent Changes

The most reliable polling strategy is to query for experiments modified since the last poll timestamp. ELabELN supports the `modified_at` field in list responses. Sort by last modified and cache the latest timestamp.

```
"""
change_poller.py
Polls ELabELN for recently modified experiments and triggers downstream actions.
"""

import os, json, time, requests
from datetime import datetime, timezone
from pathlib import Path

POLL_INTERVAL = 60 # seconds
STATE_FILE = Path('.poller_state.json')

session = requests.Session()
session.headers.update({'Authorization': os.environ['ELAB_API_KEY']})

def load_state():
    if STATE_FILE.exists():
        return json.loads(STATE_FILE.read_text())
    return {'last_check': '2020-01-01T00:00:00Z'}

def save_state(state):
    STATE_FILE.write_text(json.dumps(state))

def get_recently_modified(since_timestamp, category_id=None):
    """Fetch experiments modified after since_timestamp."""
    # Fetch a window of recently modified experiments
    # Note: ELabELN list endpoint doesn't filter by modified_at directly,
    # so we fetch recent experiments and filter client-side.
    params = {'limit': 100, 'extended': 1}
    if category_id: params['cat'] = category_id

    resp = session.get(f'{os.environ["ELAB_BASE_URL"]}/api/v2/experiments',
                       params=params)
    resp.raise_for_status()

    # Filter to those modified since our last check
    modified = [e for e in resp.json()
                 if (e.get('modified_at') or '') > since_timestamp]
    return modified

def handle_experiment_change(exp):
    """Called for each recently modified experiment. Implement your logic here."""
    fields = {}
    if exp.get('metadata'):
        try:
            fields = {k: v.get('value', '') for k, v in
                      json.loads(exp['metadata']).get('extra_fields', {}).items()}

```

```

except: pass

# Example: if status changed to Approved, sync to LIMS
if exp.get('status') == 4: # Approved
    print(f'Experiment {exp["id"]} approved - syncing to LIMS...')
    # sync_results_back_to_lims(exp['id']) # from pattern in Ch. 6

print(f'Changed: [{exp["id"]} {exp["title"][:60]}]')

def run_poller():
    print('Poller started. Ctrl+C to stop.')
    while True:
        state = load_state()
        now = datetime.now(timezone.utc).isoformat()
        try:
            changed = get_recently_modified(state['last_check'])
            for exp in changed:
                handle_experiment_change(exp)
            state['last_check'] = now
            save_state(state)
        except Exception as e:
            print(f'Poll error: {e}')
        time.sleep(POLL_INTERVAL)

if __name__ == '__main__':
    run_poller()

```

9.2 LabVia Integration — Node-RED Based Automation

LabVia is LabLynx's integration and automation platform built on Node-RED. It provides a visual flow-based programming interface and pre-built ELabELN nodes. LabVia is the recommended approach for production integrations requiring:

- Event-driven instrument data capture (triggered by file creation, serial port events, MQTT, etc.)
- Visual workflow design without Python/JS coding
- Pre-built connectors for laboratory instruments (HPLC, balances, plate readers, etc.)
- Data transformation nodes (parsing instrument output formats into ELabELN metadata)

LabVia runs on a LabServer appliance (on-premises) or as a cloud-hosted service. It connects to your ELabELN instance via the same REST API documented in this manual.

Note

For LabVia integration development, contact LabLynx at support@lablynx.com to request the LabVia node documentation and example flows specific to your instrument types. LabLynx also provides professional services for custom integration development.

10. JavaScript SDK and Browser Applications

ELabELN's API can be consumed from any JavaScript environment: Node.js backend services, browser-based custom dashboards, React/Vue/Angular frontends, and electron desktop applications. This chapter provides a reusable JavaScript module for ELabELN integration.

10.1 ELabELN JavaScript Client Module

```
/**
 * ELabELN REST API client for JavaScript (Node.js + Browser)
 * Usage: import { ELabClient } from './elab-client.js'
 */

export class ELabClient {
  constructor(baseUrl, apiKey) {
    this.baseUrl = baseUrl.replace(/\/$/, '');
    this.headers = {
      'Authorization': apiKey,
      'Content-Type': 'application/json',
      'Accept': 'application/json'
    };
  }

  _url(path) {
    return `${this.baseUrl}/api/v2/${path}`;
  }

  async _fetch(path, options = {}, params = {}) {
    const url = new URL(this._url(path));
    Object.entries(params).forEach(([k,v]) => url.searchParams.set(k, v));
    const res = await fetch(url, {
      headers: this.headers,
      ...options
    });
    if (!res.ok) {
      const body = await res.text();
      throw Object.assign(new Error(`${options.method||'GET'} ${path} failed: ${res.status}`),
        { status: res.status, body });
    }
    return res;
  }

  // --- Experiments ---
  async getExperiments(params = {}) {
    const res = await this._fetch('experiments', {}, params);
    return res.json();
  }

  async getExperiment(id) {
    const res = await this._fetch(`experiments/${id}`);
    return res.json();
  }

  async createExperiment() {
    const res = await this._fetch('experiments', { method:'POST' });
    return parseInt(res.headers.get('Location').split('/').pop(), 10);
  }
}
```

```

async updateExperiment(id, data) {
  await this._fetch(`experiments/${id}`, {
    method: 'PATCH',
    body: JSON.stringify(data)
  });
}

async addTag(id, tag) {
  await this._fetch(`experiments/${id}/tags`, {
    method: 'POST', body: JSON.stringify({tag})
  });
}

async addComment(id, comment) {
  await this._fetch(`experiments/${id}/comments`, {
    method: 'POST', body: JSON.stringify({comment})
  });
}

async linkToResource(expId, itemId) {
  await this._fetch(`experiments/${expId}/links/items`, {
    method: 'POST', body: JSON.stringify({link: itemId})
  });
}

async uploadFile(expId, file, comment='') {
  const form = new FormData();
  form.append('file', file);
  if (comment) form.append('comment', comment);
  // For file upload, override Content-Type to let browser set multipart boundary
  const uploadHeaders = Object.fromEntries(
    Object.entries(this.headers).filter(([k]) => k !== 'Content-Type')
  );
  const res = await fetch(this._url(`experiments/${expId}/uploads`), {
    method: 'POST', headers: uploadHeaders, body: form
  });
  if (!res.ok) throw new Error(`Upload failed: ${res.status}`);
  return parseInt(res.headers.get('Location').split('/').pop(), 10);
}

// --- Resources (Items) ---
async.getItems(params = {}) {
  const res = await this._fetch('items', {}, params);
  return res.json();
}

async.getItemTypes() {
  const res = await this._fetch('items_types');
  return res.json();
}

// --- Metadata helpers ---
parseMetadata(experiment) {
  try {
    const meta = JSON.parse(experiment.metadata || '{}');
    return Object.fromEntries(
      Object.entries(meta.extra_fields || {})
        .map(([k, v]) => [k, v.value || ''])
    );
  } catch { return {}; }
}

buildMetadata(fields) {

```

```

return JSON.stringify({
  extra_fields: Object.fromEntries(
    Object.entries(fields).map(([name, config], i) => [name, {
      type: config.type || 'text',
      value: String(config.value || ''),
      position: i + 1,
      ...(config.options && { options: config.options }),
      ...(config.units && { units: config.units, unit: config.unit || '' })
    }])
  )
});
}
}

// Example usage:
// const client = new ELabClient('https://yourlab.elabeln.com', 'YOUR_KEY');
// const experiments = await client.getExperiments({cat: 5, limit: 20});
// const fields = client.parseMetadata(experiments[0]);
// console.log(fields['Result (%)']);

```

10.2 Complete Node.js Integration Example

```

/**
 * plate_reader_integration.js
 * Reads plate reader CSV output and creates an ELabELN experiment.
 * Run: node plate_reader_integration.js <csv_file>
 */
import { ELabClient } from './elab-client.js';
import { readFileSync } from 'fs';
import { parse } from 'csv-parse/sync';

const client = new ELabClient(
  process.env.ELAB_BASE_URL,
  process.env.ELAB_API_KEY
);

async function importPlateReaderResults(csvFile) {
  // Parse CSV from plate reader (format: Well, OD450, Sample)
  const raw = readFileSync(csvFile, 'utf8');
  const records = parse(raw, { columns: true, skip_empty_lines: true });

  // Compute plate statistics
  const ods = records.map(r => parseFloat(r.OD450)).filter(v => !isNaN(v));
  const mean = ods.reduce((a,b)=>a+b,0) / ods.length;
  const stddev = Math.sqrt(ods.reduce((a,b)=>a+(b-mean)**2,0)/ods.length);
  const cv = (stddev / mean * 100).toFixed(1);

  // Build HTML plate layout table
  const rows = ['<table
border="1"><tr><th>Well</th><th>Sample</th><th>OD450</th></tr>'];
  for (const r of records) {
    rows.push(`<tr><td>${r.Well}</td><td>${r.Sample}</td><td>${r.OD450}</td></tr>`);
  }
  rows.push('</table>');
  const tableHtml = rows.join('');

  const today = new Date().toISOString().split('T')[0];
  const filename = csvFile.split('/').pop();

  // Create experiment
  const expId = await client.createExperiment();

```

```

// Build metadata
const metadata = client.buildMetadata({
  'Run Date': { type:'date', value: today },
  'Plate File': { type:'text', value: filename },
  'Mean OD450': { type:'number', value: mean.toFixed(4) },
  'CV (%)': { type:'number', value: cv },
  'Well Count': { type:'number', value: String(records.length) },
  'Integration Source': { type:'select', value:'Auto-Import',
                        options:['Auto-Import','Manual','Instrument-Direct'] }
});

// Populate experiment
await client.updateExperiment(expId, {
  title: `Plate Reader - ELISA - ${today} (Auto)`,
  date: today,
  category: 7, // Plate Reader / ELISA category
  body: `

## 


```

11. Complete API Endpoint Reference

This chapter provides a consolidated reference of all major API endpoints in the ELabELN REST API v2. For the full OpenAPI specification (with request/response schemas), see: <https://doc.elabftw.net/api/v2/>

11.1 All Endpoints by Resource Group

Experiments

Method	Endpoint	Description
GET	/experiments	List experiments. Query params: limit, offset, q, cat, status, owner, extended
POST	/experiments	Create empty experiment. Returns 201 + Location header.
GET	/experiments/{id}	Get full experiment detail including metadata, tags, uploads, links, steps.
PATCH	/experiments/{id}	Update experiment. Partial: only sent fields change. Returns 200.
DELETE	/experiments/{id}	Soft-delete. Returns 204.
GET	/experiments/{id}/revisions	Body text version history. Returns array of {id, created_at, body}.
GET	/experiments/{id}/changelog	Field change history. Returns array of {field, previous, next, userid, created_at}.
GET	/experiments/{id}/tags	List tags. Returns array of {tag, id}.
POST	/experiments/{id}/tags	Add tag. Body: {tag: "TagName"}. Returns 201.
DELETE	/experiments/{id}/tags/{tagId}	Remove tag. Returns 204.
GET	/experiments/{id}/uploads	List attachments. Returns array of upload objects.
POST	/experiments/{id}/uploads	Upload file. multipart/form-data: file (required), comment (optional).
DELETE	/experiments/{id}/uploads/{uploadId}	Delete attachment. Returns 204.
GET	/experiments/{id}/links	Get all links. Returns {experiments_links:[], items_links:[]}.
POST	/experiments/{id}/links/items	Link to resource. Body: {link: itemId}.
POST	/experiments/{id}/links/experiments	Link to experiment. Body: {link: expId}.
DELETE	/experiments/{id}/links/items/{itemId}	Remove resource link.

DELETE	/experiments/{id}/links/experiments/{expId}	Remove experiment link.
GET	/experiments/{id}/steps	List steps.
POST	/experiments/{id}/steps	Add step. Body: {body: "Step text", ordering: 1}.
PATCH	/experiments/{id}/steps/{stepId}	Update step. Body: {body, ordering, finished: true/false}.
DELETE	/experiments/{id}/steps/{stepId}	Delete step.
GET	/experiments/{id}/comments	List comments.
POST	/experiments/{id}/comments	Add comment. Body: {comment: "text"}.
POST	/experiments/{id}/timestamps	Apply RFC 3161 timestamp.

Resources (Items)

Method	Endpoint	Description
GET	/items	List items. Query params: limit, offset, q, cat, status, owner, extended
POST	/items	Create empty item.
GET	/items/{id}	Get full item detail.
PATCH	/items/{id}	Update item.
DELETE	/items/{id}	Delete item.
GET	/items_types	List all item type categories.
GET	/items/{id}/tags	List tags.
POST	/items/{id}/tags	Add tag.
DELETE	/items/{id}/tags/{tagId}	Remove tag.
GET	/items/{id}/uploads	List attachments.
POST	/items/{id}/uploads	Upload file.
DELETE	/items/{id}/uploads/{uploadId}	Delete attachment.
GET	/items/{id}/links	Get all links.
POST	/items/{id}/links/items	Link to another resource.
POST	/items/{id}/links/experiments	Link to experiment.
GET	/items/{id}/steps	List steps.
POST	/items/{id}/steps	Add step.
PATCH	/items/{id}/steps/{stepId}	Update/complete step.

Users, Teams, and System

Meth od	Endpoint	Description
GET	/users	List all users (SysAdmin).
GET	/users/{id}	Get user profile. Use 'me' for authenticated user.
POST	/users	Create user (SysAdmin). Body: {firstname, lastname, email, team, usergroup}.
PATCH	/users/{id}	Update user (SysAdmin or self for limited fields).
GET	/teams	List teams.
GET	/teams/{id}	Get team details.
PATCH	/teams/{id}	Update team (Admin/SysAdmin).
GET	/status	List experiment status values for current team.
GET	/tags	List all tags in current team.
PATCH	/tags/{id}	Rename a tag (Admin).
DELETE	/tags/{id}	Delete a tag (Admin).
GET	/events	List bookings. Params: start, end, item (resource ID).
POST	/events	Create booking. Body: {item, start, end, title}.
GET	/events/{id}	Get booking.
PATCH	/events/{id}	Update booking.
DELETE	/events/{id}	Cancel booking.
GET	/info	Get instance info (version, etc.). No auth required.

12. Security, Rate Limits, and Best Practices

12.1 API Key Security

Practice	Details
Never hardcode keys in source code	Always load API keys from environment variables, secrets managers (HashiCorp Vault, AWS Secrets Manager, Azure Key Vault), or secure configuration files outside the codebase.
One key per integration	Create a separate API key for each script or service. If one key is compromised, you revoke only that key without disrupting other integrations.
Use read-only keys when possible	If an integration only reads data (reporting, BI), generate a read-only API key. A compromised read-only key cannot modify or delete data.
Rotate keys periodically	Revoke and regenerate API keys at least annually, or immediately after any suspected exposure (developer departure, key committed to version control).
Monitor for unexpected usage	Review API access patterns in the ELabELN audit log. Unusual access patterns (off-hours, unexpected IP addresses, large data extraction) may indicate a compromised key.

12.2 Rate Limiting

ELabELN enforces rate limits to protect system stability. Rate limits are not publicly documented as fixed numbers because they may vary by deployment, but the following practices prevent hitting them:

- Add a `sleep()` between requests in batch loops. 0.1–0.2 seconds per request keeps volume well within limits.
- Paginate efficiently: use large page sizes (`limit=200–500`) to reduce the number of requests needed to fetch large datasets.
- Cache responses when the data is not time-critical. A status lookup that is cached for 5 minutes uses 1/300th the API calls of one that queries every second.
- Handle 429 responses with exponential backoff: wait, then retry with increasing delays.

```
import time, requests

def request_with_backoff(session, method, url, max_retries=5, **kwargs):
    """Make a request with exponential backoff on rate limit (429) errors."""
    for attempt in range(max_retries):
        resp = session.request(method, url, **kwargs)
        if resp.status_code == 429:
            wait = 2 ** attempt # 1, 2, 4, 8, 16 seconds
            retry_after = int(resp.headers.get('Retry-After', wait))
            print(f'Rate limited. Waiting {retry_after}s (attempt {attempt+1}/{max_retries})')
            time.sleep(retry_after)
        else:
            resp.raise_for_status()
            return resp
    raise Exception(f'Max retries reached for {url}')
```

12.3 Handling the Read-Modify-Write Pattern Safely

When updating metadata or other fields, always fetch the current state first to avoid race conditions and unintended overwrites. Two integrations simultaneously writing to the same experiment can produce lost updates.

- For high-frequency concurrent writes to the same experiment, serialize updates through a queue rather than parallel writes.
- Check the `modified_at` timestamp before writing: if it changed since you fetched the record, your local copy is stale and you should re-fetch before patching.

```
def safe_patch_metadata(session, base_url, exp_id, field_name, value):
    """
    Update a metadata field with optimistic concurrency check.
    Raises ValueError if the experiment was modified since we read it.
    """
    import json

    # 1. Read current state
    resp = session.get(f'{base_url}/api/v2/experiments/{exp_id}')
    resp.raise_for_status()
    exp = resp.json()
    modified_at = exp.get('modified_at', '')

    # 2. Update metadata
    meta = json.loads(exp.get('metadata') or '{}')
    meta.setdefault('extra_fields', {})[field_name] = {
        'type': 'text', 'value': str(value)
    }

    # 3. Write back (ELabELN will update modified_at server-side)
    resp2 = session.patch(
        f'{base_url}/api/v2/experiments/{exp_id}',
        json={'metadata': json.dumps(meta)}
    )
    resp2.raise_for_status()
```

12.4 Testing Your Integration

Before running any integration against a production ELabELN instance, test against a staging or development instance with synthetic data. LabLynx can provision a test instance for development use.

- Write integration tests that create, verify, and clean up test experiments. A test that leaves 500 test records in production is not a test — it is a mess.
- Use descriptive titles for test records: 'INTEGRATION_TEST — Auto-Delete — {timestamp}'. This makes test records easy to identify and clean up.
- Test all error paths: what happens when an experiment doesn't exist? When the API key is invalid? When the server is slow?

```
"""
test_integration.py
Basic integration test for ELabELN API connectivity.
Run this first to verify your credentials and connectivity.
"""
import os, json, datetime, requests

def test_api_connectivity():
```

```
API_KEY = os.environ['ELAB_API_KEY']
BASE_URL = os.environ['ELAB_BASE_URL']
session = requests.Session()
session.headers.update({'Authorization': API_KEY})

print('1. Testing authentication...')
me = session.get(f'{BASE_URL}/api/v2/users/me').json()
print(f'    Authenticated as: {me["firstname"]} {me["lastname"]}')

print('2. Creating test experiment...')
resp = session.post(f'{BASE_URL}/api/v2/experiments')
assert resp.status_code == 201, f'Expected 201, got {resp.status_code}'
exp_id = int(resp.headers['Location'].split('/')[-1])
print(f'    Created experiment ID: {exp_id}')

print('3. Populating experiment...')
ts = datetime.datetime.now().isoformat()[:19]
resp = session.patch(f'{BASE_URL}/api/v2/experiments/{exp_id}', json={
    'title': f'INTEGRATION_TEST - Auto-Delete - {ts}',
    'tags': ['integration-test']
})
assert resp.status_code == 200

print('4. Reading back experiment...')
exp = session.get(f'{BASE_URL}/api/v2/experiments/{exp_id}').json()
assert 'integration-test' in [t['tag'] for t in exp.get('tags', [])]
print(f'    Title: {exp["title"]}')

print('5. Cleaning up test experiment...')
resp = session.delete(f'{BASE_URL}/api/v2/experiments/{exp_id}')
assert resp.status_code == 204

print('All tests passed.')

if __name__ == '__main__':
    test_api_connectivity()
```

Appendix A: Environment Setup Reference

Python environment

```
# Create a virtual environment (recommended)
python -m venv elab-env
source elab-env/bin/activate # Linux/Mac
.\elab-env\Scripts\activate  # Windows

# Install dependencies
pip install requests pandas matplotlib elabapi-python

# Set environment variables
export ELAB_API_KEY='3-your-api-key-here'
export ELAB_BASE_URL='https://yourlab.elabeln.com'

# Verify: run the connectivity test
python test_integration.py
```

Node.js environment

```
# Initialize Node.js project
mkdir elab-integration && cd elab-integration
npm init -y

# Install dependencies
npm install node-fetch csv-parse dotenv

# Create .env file (do NOT commit to version control)
echo 'ELAB_API_KEY=3-your-key' >> .env
echo 'ELAB_BASE_URL=https://yourlab.elabeln.com' >> .env
echo '.env' >> .gitignore

# package.json: add "type": "module" for ES module imports
```

Required API key permissions by use case

Use Case	Required Key Type
Read experiments and resources (BI/analytics)	Read-only API key
Create experiments from instruments	Read-write API key for the integration user's account
Create and manage resources (inventory sync)	Read-write API key
Manage users and teams (admin automation)	Read-write API key on a SysAdmin account
All operations (full integration)	Read-write API key on an account with appropriate team membership

Appendix B: Metadata JSON Schema Quick Reference

```
// Complete metadata JSON schema with all field types
{
  "extra_fields": {

    // Text field
    "Analyst Name": { "type": "text", "value": "J. Smith", "position": 1 },

    // Number with units
    "Volume": {
      "type": "number", "value": "100",
      "units": ["µL", "mL", "L"], "unit": "µL", "position": 2
    },

    // Date
    "Expiry Date": { "type": "date", "value": "2026-12-31", "position": 3 },

    // Datetime
    "Run Start": { "type": "datetime-local", "value": "2026-06-01T09:00",
      "position": 4 },

    // Select (dropdown)
    "Sample Type": {
      "type": "select",
      "value": "Blood",
      "options": ["Blood", "Urine", "CSF", "Tissue"],
      "position": 5
    },

    // Multi-select
    "Tests Requested": {
      "type": "select",
      "value": "HPLC",
      "options": ["HPLC", "GC-MS", "LC-MS", "NMR"],
      "allow_multi_values": true,
      "position": 6
    },

    // Radio
    "Priority": {
      "type": "radio",
      "value": "Routine",
      "options": ["STAT", "Priority", "Routine"],
      "position": 7
    },

    // Checkbox
    "Controls Included": { "type": "checkbox", "value": "on", "position": 8 },
    // checkbox: "on" = checked, "" = unchecked

    // URL
    "Raw Data": { "type": "url", "value": "https://labdrive.../file.raw",
      "position": 9 },

    // Email
    "Contact": { "type": "email", "value": "analyst@lab.com", "position": 10 },

    // Field in a named group
    "Batch Number": {
```

```
    "type": "text", "value": "BATCH-001",  
    "group_id": 1, "position": 11  
  },  
  "elabftw": {  
    "display_main_text": true,  
    "extra_fields_groups": [  
      { "id": 1, "name": "Production Info" },  
      { "id": 2, "name": "QA Results" }  
    ]  
  }  
}
```



ELabELN Developer Manual

Version 1.0 | June 2026

LabLynx, Inc. | www.lablynx.com | support@lablynx.com | www.elabeln.com

ELabELN is powered by LabLynx and built on the eLabFTW open-source platform. API specification:
<https://doc.elabftw.net/api/v2/>

© 2026 LabLynx, Inc. All rights reserved.